# CS - F372 OPERATING SYSTEMS

## SECOND SEMESTER 2024-2025

## ASSIGNMENT 2: PORT MANAGEMENT SYSTEM
## MAX. MARKS = 45

## DEADLINE: 2:00 AM, 14/04/2025

**Note: This document contains the problem statement, assignment constraints, hints, submission guidelines, late submission policy, plagiarism policy, and demo guidelines. Please go through the entire document thoroughly.**

## Problem Statement

Captain Jack Sparrow has retired from his swashbuckling pirate life and taken up a respectable – though equally challenging – role as the Portmaster of Tortuga, now a bustling cargo port. The infamous pirate haven has transformed into a thriving trade hub, where ships laden with valuable cargo, dock to unload their goods and prepare for departure. Jack may have traded his compass for a clipboard, but managing the busy docks of Tortuga is proving to be no less tricky.

At Tortuga, there are a specific number of docks, and each dock (each dock can only be occupied by one ship at a time) is equipped with cranes that can only lift up to a certain weight at a time. The port sees a constant flow of **cargo ships** – both incoming and outgoing. The incoming ships can be of two types: regular and emergency. Each incoming ship must be docked at a particular dock, unloaded, and then undocked to free that dock. Similarly, every outgoing ship must be docked at a particular dock, loaded with cargo, and then undocked. Each ship has a category associated with it. Similarly, each dock has a category associated with it. Each category is a non-zero integer value, and a dock of category X has X cranes present at it. A ship can only be docked at a dock that has a category value >= the ship's category value. Every ship carries cargo boxes of varying weights, and the cranes must be carefully allocated to ensure smooth and efficient unloading without exceeding their weight limits.

Jack quickly realizes that it's not just about unloading and loading cargo – it's about managing the delicate balance between incoming and outgoing ships to prevent overcrowding and chaos. A backlog at the docks can lead to **unhappy merchants, frustrated captains, and tense negotiations** – none of which will bode well for Tortuga's hard-earned reputation as a reliable port.

There is another catch, before undocking, all the ships have to guess a secure radio channel frequency that connects them to Jack and notifies him before they undock. As the port starts serving more and more ships, Jack must carefully manage the flow to avoid overcrowding and chaos.

Now, Captain Jack Sparrow is counting on your help. You and your team must develop a system that can handle ship scheduling, optimize crane allocation, and manage cargo unloading and loading efficiently – all the while prioritizing emergency docking requests. It's a delicate balance to keep Tortuga's port running smoothly, prevent delays, and maintain order. After all, brawls, blockades, and brewing tensions are the last things this thriving trade hub needs.

Managing a port may not be as thrilling as plundering treasure on the high seas, but with your skills – and a bit of pirate ingenuity – you can help Jack keep Tortuga in shipshape and maintain its newfound reputation as the most efficient cargo port on the high seas!

# System Overview

**Note:** *The term "scheduler program" refers to the program to be developed by you and your group. In the rest of this document, the terms scheduler program and scheduler process will be used interchangeably.*

The system simulates a port scheduling and management system, where the **scheduler program** must handle requests from ships to dock or undock. The ship requests are simulated through a **validation module** (provided to you). The communication between the validation module and the scheduler program is done using a **message queue** and a **shared memory segment**.

The validation also spawns a set of solver processes. A message queue is used between each of these solver processes and the scheduler program (the functioning of the solver processes is described later).

In the rest of the document, the message queue between the scheduler program and the validation module is referred to as the **main message queue**, and the message queues between the solver processes and the scheduler program are referred to as the **solver process message queues**.

## Constraints

- 1 <= Number of Docks <= 30
- 1 <= Dock Category <= 25
- 1 <= Ship Category <= 25
- 1 <= Number of regular incoming ships <= 500
- 1 <= Number of emergency incoming ships <= 100
- 1 <= Number of outgoing ships <= 500
- 1 <= Capacity of a Crane <= 30
- 1 <= Cargo Count of a Ship <= 250
- 2 <= Number of Solvers <= 8
- Maximum New Ship Requests arriving in a given timestep: 100

## Input

Your program needs to read the file `testcase_X/input.txt`, where `X` represents the test case number. This file contains the following information in sequence:

1. **Shared memory key** – key for the shared memory segment between validation and the scheduler program.
2. **Main message queue key** – key for the main message queue between validation and the scheduler program.
3. Number of available solvers (`m`)
4. m lines containing the keys for the message queues between the solver processes and the scheduler program
5. Number of docks (`n`)
6. n lines containing the information for each dock. Each of these lines has X + 1 space separated integers (where X is the category of the dock). The first integer is the category of the dock (X), and the following X integers are the weight capacities of the X cranes available at the dock.

An sample input.txt is given below

---

```
12344135
67889502
4
83111411
94739478
28366493
58636462
3
5 2 3 1 4 2
4 2 1 3 1
3 3 3 4
```

---

### Explanation of the sample input.txt:

- **Shared Memory Key**: `12344135`
- **Main Message Queue Key**: `67889502`
- **Number of Solvers**: `4` (denoted by $m = 4$).
    - `83111411` – Message queue key for Solver 1.
    - `94739478` – Message queue key for Solver 2.
    - `28366493` – Message queue key for Solver 3.
    - `58636462` – Message queue key for Solver 4.

- **Number of Docks**: 3 (denoted by *n = 3*), with the next 3 lines providing dock details.

1. **First Dock** :
    - Dock Category: 5
    - Crane Capacities: 2 3 1 4 2

2. **Second Dock**:
    - Dock Category: 4
    - Crane Capacities: 2 1 3 1

3. **Third Dock** :
    - Dock Category: 3
    - Crane Capacities: 3 3 4

***Note**: Dock IDs are **assigned sequentially**, starting from 0, based on the order they appear in **input.txt**. The variable dockId used in the remaining document takes these values. In the above example, the First Dock (of category 5) has dockId 0, the Second Dock (of category 4) has dockId 1, and the Third Dock (of category 3) has dockId 2.*

*Similarly, Crane IDs start from 0 for **each dock** and are assigned sequentially. The variable craneId used in the remaining document takes these values. In the above example, for the First Dock, the first crane (of capacity 2) has craneId 0, the second crane (of capacity 3) has craneId 1, and so on. Similarly, for the Second Dock, the first crane (of capacity 2) has craneId 0, the second crane (of capacity 1) has craneId 1, and so on.*

## Validation Module

The validation module interacts with the scheduler program and performs the following key tasks:

### 1. Tracking a global timestep

The functionality of the entire system is based on a global timestep. This value begins at 1 and increases by 1 after a set of actions performed by the scheduler program (details of these actions are provided later).

### 2. Sending New Ship Requests

At the beginning of each timestep, the validation sends a message of **mtype 1** to the main message queue (intended for the scheduler process), indicating the number of new ship requests for the current timestep using the following struct **MessageStruct**.

```
typedef struct MessageStruct {

        long mtype;
        int timestep;
        int shipId;
        int direction;
        int dockId;
        int cargoId;
        int isFinished;
        union {
            int numShipRequests;
            int craneId;
        };
    } MessageStruct;
```

In this message, validation will set the mtype to 1, the timestep field to the current timestep, and the **numShipRequests** field to the number of new ship requests. The remaining fields of the struct will be undefined **in this message** and hence should be ignored. These fields will be used for subsequent communication between the validation module and the scheduler program (described later).

The details of the new ship requests will be present in the shared memory segment between the validation module and the scheduler program. The shared memory segment uses the following struct **MainSharedMemory**.

```
typedef struct MainSharedMemory{
    char authStrings[MAX_DOCKS][MAX_AUTH_STRING_LEN];
    ShipRequest newShipRequests[MAX_NEW_REQUESTS];
} MainSharedMemory;
```

The new ship request details will be present in the indices **0 to numShipRequests - 1** of the **newShipRequests** array. The authStrings array of the shared memory should be ignored while reading the new ship requests. **newShipRequests** is an array of type **ShipRequest,** which is defined as follows:

```
typedef struct ShipRequest{
    int shipId;
    int timestep;
    int category;
    int direction;
    int emergency;
    int waitingTime;
    int numCargo;
```

```
    int cargo[MAX_CARGO_COUNT];
} ShipRequest;
```

Below is a detailed explanation of each data field in the *ShipRequest* struct:

- **int shipId:** Represents the unique identifier for the ship, used to track and manage individual ships.
- **int timestep:** Indicates the timestep at which the new ship request was sent.
- **int category:** Indicates the category of the ship. The ship can **only be assigned to a dock with a category greater than or equal to this value**. This is referred to as the **category constraint.**
- **int direction:** Specifies whether the ship is incoming or outgoing:
    - `1` – Incoming
    - `-1` – Outgoing

  **Important Note:** The combination of the *shipId* field and the *direction* field uniquely identifies a ship. This means that an incoming and outgoing ship can have the same *shipId* value, but they will not have the same direction value. Two incoming ships, however, cannot have the same *shipId* value (but they will have the same *direction* value). Similarly, two outgoing ships cannot have the same *shipId* value.

- **int emergency:** Indicates whether the ship is in an emergency state (applicable only to incoming ships):
    - `0` – Regular ship.
    - `1` – Emergency ship (requires priority handling).
      **Note:** This field should be ignored for outgoing ships (i.e., when `direction = -1`).
- **int waitingTime:** All **regular incoming ships** have a waiting time. If they are not assigned a dock within this time, the ship will leave the port and return later as a new ship request. This waitingTime field specifies this time window within which the ship must be assigned a dock. The ship can be docked anytime from its arrival timestep until `arrival timestep + waitingTime`. If it is not assigned a dock by this time, the ship will leave and return later as a new request. Thus, from timestep value = `arrival timestep + waitingTime + 1`, you cannot assign a dock to this ship until it returns to the port. When the ship returns to the port, validation will set it as a new ship request in the shared memory segment in the same way it does for new ships. If the returning ship is again not assigned a port

within the waiting time, it will again leave the port and come back at a later time. This sequence of events will continue until the scheduler services that ship.

**Note:** This field is **not applicable** for **emergency** and **outgoing** ships.

- **int numCargo:** Represents the number of cargo items the ship is carrying. This value indicates how many cargo weights are stored in the `cargo` array.
- **int cargo[MAX_CARGO_COUNT]:** An array storing the weight of each cargo item. The array contains `numCargo` elements, and each element represents the weight of a specific cargo item on the ship. **Cargo IDs are assigned sequentially in the order they appear, starting with 0. The variable *cargoId* used in the remaining document takes these values. The first cargo item in the array cargo has *cargoId* 0, the second item has *cargoId* 1, and so on.**

**Note:** In this document, the term **"leaving"** refers to a ship going away from the port when it has not been assigned a dock within its waiting time. Such a ship will return later as a new request, retaining the same parameters except the updated timestep generated by the validation module. This is distinct from **"undocking,"** which specifically means freeing an occupied dock after the ship's request has been successfully "**serviced"** (i.e., the ship is assigned a suitable dock and all of its cargo is moved). In the case of incoming ships, this happens after they have docked and fully unloaded their cargo, and for outgoing ships, after they have docked and finished loading cargo. The term "**moving**" cargo refers to either loading or unloading cargo.

**Note:** When a dock is assigned to a ship (any type), that dock gets blocked and cannot service other ships unless the currently occupying ship undocks. A dock which is occupied by a ship is referred to as an **occupied dock.** A dock which is not occupied by any ship is referred to as a **free dock.**

The **life cycles** of the different types of ship requests are described below:

### Life Cycle of an Incoming Regular Ship

1. **Dock Request**: The ship requests a dock.
2. **Dock Assignment**:
    - If assigned a dock within the specified waiting time, it proceeds to dock
    - If a dock is not assigned within the waiting time, the ship leaves and returns later as a new request.

3. **Cargo Unloading**: Once docked, the ship unloads its cargo.

4. **Undocking**: After unloading all its cargo, the ship undocks.

### Life Cycle of an Incoming Emergency Ship

The emergency ship follows the same process as a regular incoming ship, with one key difference: it has no waiting time and must be serviced as soon as possible. We enlist the requirements to handle emergency ships in detail later in the document.

Note: No emergency ship can pre-empt a regular ship (either incoming or outgoing) already docked at a dock and being serviced

### Life Cycle of an Outgoing Ship

1. **Dock Request**: The ship requests a dock.

2. **Dock Assignment**: When a dock is assigned, the ship begins loading cargo.

3. **Undocking**: After all the cargo has been loaded, the ship undocks.

## 3. Validating various actions performed by the scheduler program

The validation module tracks the current status of the port (the status of all ships, the free docks, occupied docks, ships which have been serviced, etc.) and ensures that the actions performed by the scheduler program adhere to the rules of the port. Further, it handles the solver processes that are required to undock a ship. (All these actions of the scheduler program and the rules they must adhere to are described in the next section.)

# The Scheduler Program

Your task is to design and implement the ship scheduler program that efficiently manages ship arrivals, dock allocation, servicing, cargo unloading, cargo loading, and ship undocking. It must ensure to adhere to the rules of the port and handle all the ship requests provided by the validation module. Every action that is performed by the scheduler program – docking a ship, loading/unloading cargo, or undocking a ship – must be communicated by the scheduler program to the validation module. Further, to undock a ship, the scheduler program must guess a radio frequency (string) with the assistance of the solver processes. Any action performed by the scheduler program that violates the rules listed below will fail the testcase, and the validation program will terminate immediately.

We begin by describing each action to be performed by the scheduler program and how it should communicate the same with the validation module.

1. **Setting up the various IPC mechanisms**

   The scheduler program must establish the required IPC mechanisms for communication with the validation module and the solver processes. These include establishing a connection with the main message queue between the validation module and the scheduler program, establishing a connection with the message queues between the solver processes and the scheduler program, and connecting to the shared memory segment between the validation module and the scheduler program.

2. **Polling for Requests from Validation at Each Timestep**

   At the beginning of each timestep, your program must read messages of `mtype = 1` from the main message queue (structured as `MessageStruct`) to check for any new ship requests. In this message, validation will set the mtype to 1, the timestep field to the current timestep, and the **numShipRequests** field to the number of new ship requests. The remaining fields of the struct will be undefined **in this message** and hence should be ignored. The new ship request details will be present in the indices **0 to numShipRequests - 1** of the **newShipRequests** array in the shared memory segment between validation and the scheduler program. The details of how to read the newShipRequest items have already been outlined in the validation module section above.

3. **Docking a Ship**

   The scheduler program must dock a ship before loading/unloading cargo for it. At a particular timestep, the scheduler program may choose to dock a ship at a free (unoccupied) dock with a category greater than or equal to the ship's category (or it may choose to postpone this docking to a later timestep). To dock a ship, the scheduler program must use the struct `MessageStruct` and send a message of mtype 2 to the validation process using the main message queue. In this message, the scheduler program must set the *dockId* variable of the struct to the ID of the dock at which it wishes to dock the ship. It must also set the *shipId* field and the *direction* field of the struct to identify which ship is being docked. The value of the shipId and direction fields should be set to their corresponding values received from the **newShipRequests** array in the shared memory for this ship.

4. **Loading/Unloading Cargo**

   Once a ship has been docked, the cargo must be unloaded from it if it is an incoming ship, or loaded onto the ship if it is an outgoing ship. To unload/load a cargo item, the scheduler program must use a crane available at the dock where the ship is docked, and the crane's

capacity must be greater than or equal to the weight of the cargo item. At each timestep, a crane can only be used to move one cargo item. To load/unload a cargo item, the scheduler program must use the struct `MessageStruct` and send a message of mtype 4 to the validation process using the main message queue. In this message, the scheduler program must set the *dockId* variable of the struct to the ID of the dock at which the ship is docked. It must also set the *shipId* and *direction* fields of the message in the same way they were set to dock the ship. Apart from these, the scheduler program must also set the *cargoId* field and the *craneId* field in the message. Determining these IDs has been described above in the Input section.

## 5. Undocking Ships

Once all the cargo has been loaded/unloaded from a ship, it can then be undocked to free the dock it was occupying. However, to undock a ship, the scheduler must first guess a radio frequency string correctly. The length of this radio frequency string is equal to the value: **Timestep at which the last piece of cargo was moved from the ship minus Timestep at which the ship was docked**. This string has the following properties:

- Each character of the string can be `5`, `6`, `7`, `8`, `9`, or `'.'`
- The first and last characters cannot be `'.'`

To guess this string, the scheduler program must communicate with the solver processes. For this communication, the following structs, SolverRequest and SolverResponse, must be used

```
typedef struct SolverRequest{
    long mtype;
    int dockId;
    char authStringGuess[MAX_AUTH_STRING_LEN];
} SolverRequest;

typedef struct SolverResponse{
    long mtype;
    int guessIsCorrect;
} SolverResponse;
```

To guess the string for a dock X, the scheduler program must do the following:

1. First, the scheduler process must inform each solver it wants to use which dock it is guessing the frequency string for. To do this, the scheduler process must send a message using struct `SolverRequest` of **mtype 1** to each solver process it intends

to use, using the message queue for that solver. In this message, the *dockId* field must be set to the ID of the dock that the scheduler process is guessing for. The authStringGuess field should be ignored for this message. The solver process **will NOT** send a message in response to this message.

2. Once the dock has been set, the scheduler process can start guessing the frequency string. To make a guess, the scheduler process must send a message using struct `SolverRequest` of **mtype 2** to a solver process. In this message, the *authStringGuess* field must be set to the string the scheduler process is guessing. In response to this message, the solver process will send a message in response using the struct `SolverResponse` of **mtype 3.** In this message, the *guessIsCorrect* field of the SolverResponse struct will be set to one of the following 3 values:
    - 0: Indicating that the guess is incorrect for this dock
    - 1: Indicating that the guess is correct for this dock
    - -1: Indicating that either the dock being guessed for has not been set, or all the cargo for the ship at the dock has not been moved, or there is no ship at this dock.
3. Once the scheduler process has guessed the correct string, it must put this string in the *authStrings* field of the shared memory segment between the validation and the scheduler process.
4. After this, the scheduler process needs to send a message to validation using the struct `MessageStruct` of **mtype 3**, using the main message queue. In this message, the *dockId* field should be set to the ID of the dock from which the ship is undocking. The *shipId* and *direction* fields should also be set to indicate which ship is undocking.

If the guessed string is correct, the ship will undock and the dock will be freed. Otherwise, the testcase will fail and validation will terminate immediately.

## 6. Updating the Timestep

Once the scheduler program has performed all the actions it wishes to in a particular timestep, it must tell the validation module to proceed to the next timestep. This is done by sending a message to the validation module using the struct `MessageStruct` with a mtype of 5. In this message, only the mtype field is required, and the other fields should be ignored. After this message, validation will update the timestep and send a new message containing the number of new ship requests for the updated timestep. The rules for the set of actions that can be performed in a given timestep are given after the Termination section.

## 7. Termination

Once all the ship requests have been serviced, validation will inform the scheduler process that the test case is complete. To check this, in the initial message sent by the validation module indicating the number of new ship requests at a given timestep, check if the

*isFinished* field in `MessageStruct` has been set to `1`. If *isFinished* = `1`, it indicates that all ship requests have been serviced, and the test case has concluded.

### Rules Regarding Docking or Undocking a Ship and Loading/Unloading Cargo

- If a new ship request arrives at timestep T, the ship can be assigned to any free dock with dock's category >= the ship's category at any timestep from timestep T. If the ship is a regular incoming ship, it will have a waiting time of X. In this case, the ship can be assigned a dock at any time step from T to T + X (both inclusive). If it is not assigned a dock in this period, the ship will leave and return later as a new request at time step T'. Again, this ship can be assigned a dock at any timestep from T' to T' + X (both inclusive). If it is not assigned a dock in this period, the ship will leave again and return later and so on. For emergency incoming ships and outgoing ships, there is no waiting time, and hence they can be assigned a dock at any time step from T.
- If a ship is docked at time step T, no cargo can be loaded/unloaded from the ship until time step T + 1. Trying to load/unload cargo from the ship at time step T will fail the testcase, and validation will terminate immediately.
- If the last cargo item was loaded/unloaded from a ship at time step T, the ship cannot be undocked until time step T + 1. Attempting to undock the ship at time step T will fail the testcase, and validation will terminate immediately.
- If a ship is undocked from a dock D at time step T, no new ship can be docked at dock D until time step T + 1. Attempting to dock a ship at dock D at time step T will fail the testcase, and validation will terminate immediately.

### Rules for Servicing Emergency Ships

- An emergency ship must be serviced as soon as possible by a free dock that has a category >= the ship's category. An emergency ship **cannot** preempt a ship (of any type) that is currently occupying a particular dock.
- At the beginning of each time step T, if there are Z free docks, and Y emergency ships (including those that came as new requests at time step T), then the scheduler program must ensure that the maximum possible number of these emergency ships are assigned to the free docks subject to the category constraint. If the number of emergency ships assigned at a particular time step is less than this maximum possible number, the testcase will fail and validation will terminate immediately.

## Steps to Test

1. **Open Two Terminals:**

    - Please note that the executable file for the validation process has been provided (validation.out). You may have to execute the command chmod 777 validation.out before running the file.

- Run the validation program in one terminal with `X` as a command-line input, where `X` denotes the test case number. Use the command ./validation.out X, where X is the test case number.
- Run your scheduler program in the second terminal. Use the command ./scheduler.out X, where X is the test case number.

  **Note**: The validation program must be run first.

2. **Directory Structure:**

   - Ensure that the `validation` program and the `testcase_X` folders are present in the same parent directory. This setup is necessary for relative paths to function correctly.

3. **File Access Rules:**

   - Your program **must only read the `input.txt`** file located inside the `testcase_X` folder.
   - The other files are intended exclusively for the `validation` program and **must not be accessed by your program**.

4. **Demo Warning:**

   - During the demo, the format of the files (other than `input.txt`) may be modified.
   - If your program attempts to read any restricted file directly, it will not execute properly, and no reconsiderations will be made.

## Sample Test Case Constraints

There are 6 test cases provided along with the assignment statement. Your scheduler program is expected to pass these test cases within the specified simulated timestep thresholds (timesteps given by scheduler <= threshold)-
TestCase 1 - 27 timesteps
TestCase 2 - 186 timesteps
TestCase 3 - 232 timesteps
TestCase 4 - 291 timesteps
TestCase 5 - 512 timesteps
TestCase 6 - 600 timesteps

Along with the timestep threshold your program is expected to run within **6 minutes (in real time)**. **During the demo, if your program does not terminate with the correct output within 6 minutes, the evaluator will forcibly terminate the execution and the test case will be considered as failed.**

**However, if your program terminates within 6 minutes but takes even 1 timestep more than the designated number of timesteps, the test case will be considered as failed.**
**The duration of 6 minutes has been determined by running the sample test case on a machine running with Ubuntu 24.04 LTS , i7-8700 processor and 16 GB RAM.**
The test case is considered passed completely when the scheduler solves the test case correctly and within both the timestep and real time thresholds.

## Implementation Guidelines and Constraints:

- **Source File:**
  - Implementation should only be done using the C language. No other programming language is allowed.
  - Note that you need to submit a single POSIX-compliant C program which can be named as scheduler.c
  - No other .c file should be submitted.
- **Using validation.out:**
  - If you are an Intel/AMD/Older Mac (x64) user, use this validation.out file.
  - If you are a Mac M1 (AArch64) user, use this validation.out file.

- **Error Handling:**
  - Perform proper error handling for all system calls.
  - Failure to handle errors appropriately will result in mark deductions.

- **File Specifications:**
  - All .txt files mentioned are ASCII (text-only) files.

- **Inter-Process Communication (IPC):**
  - Only the specified IPC mechanisms—**shared memory** and **message queues**—are to be used.
  - The use of any other IPC mechanisms will result in mark deductions.

- **Input Restrictions:**
  - No additional inputs are allowed beyond those specified in the problem statement.
  - You should not read any file other than `input.txt`
  - Any deviation from this will lead to mark deductions.

- **Synchronization and Concurrent Processing:**
  - Use appropriate synchronization constructs to avoid race conditions.
  - Use concurrent processing if you feel it is required.

- **Adherence to Constraints:**
  - Any violation of the constraints mentioned above will lead to deductions in marks.

- **Use of unstructured programming constructs:**
  - You are not allowed to use unstructured programming constructs like goto in any part of your code. Marks will be deducted if you use any such constructs.

## Submission Guidelines:

● The submitted code should be a POSIX-compliant C program. No other programming language should be used.

● All codes should run on Ubuntu 22.04 or 24.04 systems. You can be asked to demo the application on either system variant.

● Submissions are to be done through the Google Form:
**https://docs.google.com/forms/d/1QuPn72j5YfyVqmlhnasD2uk_G17WiJFWWs-PlgIFcAM/preview**

● There should be only one submission per group.

● Each group should submit a zipped file containing all the relevant C program(s) and a text file containing the correct names and IDs of all the group members. The names and IDs should be written in uppercase letters only. The zipped file should be named GroupX_A2 (X stands for the group number). You will be notified of your group number after a few days.

● If there are multiple submissions from a single group, any one of the submissions will be considered randomly.

● Your group composition for Assignment 2 has been frozen now. You cannot add or drop any group member now.

● No extensions will be granted beyond the given deadline.

● Do not attempt any last-minute submissions. You need to be mindful of situations such as laptops not working at the last moment, Google Form becoming unresponsive, etc.

## Late Submission Policy:

● The Google form will accept submissions beyond the given deadline.

● SUBMITTING AFTER THE DEADLINE WILL INCUR A LATE PENALTY.

● For every 20 minutes of delay, there will be a late penalty of 0.25 marks. However, no distinction will be made within that 20-minute window. For eg., The deadline is 2:00 AM on 14/04/2025. If a group submits between 2:01 AM and 2:20 AM (inclusive), then that group will lose 0.25 marks. If the submission is made at 2:10 AM, then also the penalty will be 0.25 marks. However, if the submission is made at 2:21 AM, then 0.5 marks will be deducted.

● The submission will be summarily closed after 48 hours from the deadline. No submission will be allowed after 48 hours from the deadline.

● It is your discretion what you want to do – take a penalty and correct some last minute error or submit a partially correct implementation (which will lead to marks deduction).

● For calculating the late penalty, the date and timestamp of the submission via the Google form will be considered. No arguments will be entertained in this regard.

## Plagiarism Policy:

● All submissions will be checked for plagiarism.

● Lifting code/code snippets from the Internet is plagiarism. Taking and submitting the code of another group(s) is also plagiarism. However, plagiarism does not imply discussions and exchange of thoughts and ideas.

● All cases of plagiarism will result in awarding a hefty penalty. Groups found guilty of plagiarism may be awarded zero.

● All groups found involved in plagiarism, directly or indirectly will be penalized.

● The entire group will be penalized irrespective of the number of group members involved in code exchange and consequently plagiarism. So, each member should ensure proper group coordination.

● The course team will not be responsible for any kind of intellectual property theft. So, if anyone is lifting your code from your laptop, that is completely your responsibility. Please remember that it is not the duty of the course team to investigate cases of plagiarism and figure out who is guilty and who is innocent.

● **PLEASE BE CAREFUL ABOUT SHARING CODE AMONG YOUR GROUP MEMBERS VIA ANY ONLINE CODE REPOSITORIES. BE CAREFUL ABOUT THE PERMISSION LEVELS (LIKE PUBLIC OR PRIVATE). INTELLECTUAL PROPERTY THEFT MAY ALSO HAPPEN VIA PUBLICLY SHARED CODE REPOSITORIES.**

● **SUBMITTING CODES CREATED BY PROMPTING LANGUAGE MODELS IS STRICTLY PROHIBITED. IF SUBMITTED CODES ARE FOUND TO BE GENERATED BY A LANGUAGE MODEL, THE ENTIRE GROUP WILL DEFINITELY BE PENALIZED AND MAY BE AWARDED ZERO.**

## Demo Guidelines:

● The assignment also consists of a demo component to evaluate each student's effort and level of understanding of the implementation and the associated concepts.

● The demos will be conducted in either the I-block labs or D-block labs. Therefore, the codes submitted through the Google Form will be tested on the lab machines.

● No group will be allowed to give the demo on their laptop.

● The codes should run on Ubuntu 22.04 or Ubuntu 24.04.

● All group members should be present during the demo.

● Any absent group member will be awarded zero.

● The demos will be conducted in person. The demos will not be rescheduled.

● Though this is a group assignment, each group member should have full knowledge of the complete implementation. During the demo, questions may be asked from any aspect of the assignment.

● **Demos will be conducted on 19th April and 20th April 2025. Note that demos can be conducted on any one or both days. So, you need to be available on campus for both days. You need to book your demo slots as per the availability of your entire group. If any evaluation component of some other course gets scheduled on these dates, you will need to book your demo slot to avoid any clashes**.

● Demo slots will be made available in due time. You need to book your demo slots as per the availability of your entire group.

● The code submitted through Google Forms will be used for the demo. No other version of the codes will be considered.

● Each group member will be evaluated based on their overall understanding and effort. A group assignment does not imply that each and every member of a group will be awarded the same marks.

● Any form of bargaining for marks with the evaluators will not be tolerated during the demo.